A reliable Turing machine

Ilir Çapuni Peter Gács

August 14, 2013

Abstract

The title says it.

1 Introduction

1.1 To be written

1.2 Turing machines

Our contribution uses one of the standard definitions of a Turing machine.

A Turing machine M is defined by a tuple

$$(\Gamma, \Sigma, \delta, q_{\text{start}}, F).$$

Here, Γ is a finite set of ***states***, Σ is a finite alphabet, and

$$\delta\colon \Sigma \times \Gamma \to \Sigma \times \Gamma \times \{-1, 0, +1\}$$

is the transition function. The tape alphabet Σ contains at least the distinguished symbols $\sqcup, 0, 1$ where $\sqcup$ is called the ***blank symbol***. The state q_{start} is called the ***starting state***, and there is a set F of ***final states***.

The tape is blank at all but finitely many positions.

A ***configuration*** is a tuple

$$(q, A, h),$$

where $q \in \Gamma$ is the ***current state***, $h \in \mathbb{Z}$ is the current ***head position***, or ***observed cell***, and $A \in \Sigma^{\mathbb{Z}}$ is the ***tape content***: at position p, the tape contains the symbol $A[p]$. If $C = (q, A, h)$ is a configuration then we will write

$$C.\text{state} = q, \quad C.\text{pos} = h, \quad C.\text{tape} = A.$$

Here, A is also called the ***tape configuration***. The cell at position h is the ***current cell***. Though the tape alphabet may contain non-binary symbols, we will restrict input and output to binary.

The transition function δ tells us how to compute the next configuration from the present one. When the machine is in a state q, at tape position h, and observes tape cell with content a, then denoting

$$(a', q', j) = \delta(a, q),$$

it will change the state to q', change the tape content at position h to a', and move to tape position to $h + j$. For $q \in F$ we have $a' = a$, $q' \in F$.

Definition 1.1 (Fault)**.** We say that a ***fault*** occurs at time t if the output (a', q', j) of the transition function at this time is replaced with some other value (which is then used to compute the next configuration). ⌟

1.3 Codes, and the result

For fault-tolerant computation, some redundant coding of the information is needed.

Definition 1.2 (Codes)**.** Let Σ_1, Σ_2 be two finite alphabets. A ***block code*** is given by a positive integer Q—called the ***block size***—and a pair of functions

$$\psi_* : \Sigma_2 \to \Sigma_1^Q, \quad \psi^* : \Sigma_1^Q \to \Sigma_2$$

with the property $\psi^*(\psi_*(x)) = x$. It is extended to strings by encoding each letter individually: $\psi_*(x_1, \ldots, x_n) = \psi_*(x_1) \cdots \psi_*(x_n)$. ⌟

For ease of spelling out a result, let us consider only computations whose outcome is a single symbol, at tape position 0.

Theorem 1. *There is a Turing machine M_1 with a function $a \mapsto a.output$ defined on its alphabet, such that for any Turing machine G with alphabet Σ and state set Γ there are $0 \leqslant \varepsilon < 1$ and $\alpha_1, \alpha_2 > 0$ with the following property.*

For each input length $n = |x|$ a block code (ϕ_*, ϕ^*) of blocksize $Q = O((\log n)^{\alpha_1})$ can be constructed such that the following holds.

Let M_1 start its work from its initial state, and the initial tape configuration $\xi = (q_{\text{start}}, \phi_*(x), 0)$. Assume further that during its operation, faults occur independently at random with probabilities $\leqslant \varepsilon$.

Suppose that on input x machine G reaches a final state at time t and writes value y at position 0 of the tape. Then denoting by $\eta(u)$ the configuration machine M_1 at time u, at any time t' after

$$t \cdot (\log t)^{\alpha_2},$$

we have $\eta(t')$.tape[0].output $= y$ with probability at least $1 - O(\varepsilon)$.

We emphasize that the actual code ϕ of the construction will depend on n only in a simple way: it will be the "concatenation" of one and the same fixed-size code with itself, $O(\log \log n)$ times.

2 Overview of the construction

A Turing machine that simulates "reliably" any other Turing machine even when it is subjected to isolated bursts of faults of constant size, is given in [1]. By ***reliably*** we mean that the simulated computation can be decoded from the history of the simulating machine despite occasional damages.

2.1 Isolated bursts of faults

Let us give a brief overview of a machine M_1 that can withstand isolated bursts of faults, as most of its construction will be reused in the probabilistic setting.

Let us break up the task of error correction into several problems to be solved. The solution of one problem gives rise to another problem one, but the process converges.

Redundant information. The tape information of the simulated Turing machine will be stored in a redundant form, more precisely in the form of a block code.

Redundant processing. The block code will be decoded, the retrieved information will be processed, and the result recorded. To carry out all this in a way that limits the propagation of faults, the tape will be split into tracks that can be handled separately, and the major processing steps will be carried out three times within one work period.

Local repair. All the above process must be able to recover from a local burst of faults. For this, it is organized into a rigid, locally checkable structure with the help of local addresses, and some other tools like sweeps and zigging. The major tool of local correction, the local recovery procedure, turns out to be the most complex part of the construction.

Disturbed local repair. A careful organization of the recovery procedure makes sure that even if a new burst interrupts it (or jumps into its middle), soon one or two new invocations of it will finish the job (whenever needed).

Here is some more detail.

Each tape cell of the simulated machine M_2 will be represented by a block of size Q of the simulating machine M_1 called a ***colony***. Each step of M_2 will be simulated by a computation of M_1 called a ***work period***. During this time, the head of M_1 makes a number of sweeps over the current colony, decodes the represented cell symbol and state, computes the new state, and transfers the necessary information to the neighbor colony that corresponds to the new position of the head of M_2.

In order to protect information from the propagation of errors, the tape of M_1 is subdivided into ***tracks***: each track corresponds to a ***field*** of a cell symbol of M_1 viewed as a data record. Each stage of computation will be repeated three times. The results will be stored in separate tracks, and a final cell-by-cell majority vote will recover the result of the work period from them.

All this organization is controlled by a few key fields, for example a field called *cAddr* showing the position of each cell in the colony, and a field *cSw* showing the last sweep of the computation (along with its direction) that has been performed already. The technically most challenging part of the construction is the protection of this control information from bursts.

For example, a burst can reverse the head in the middle of a sweep. Our goal is that such structural disruptions be discovered locally, so we cannot allow the head to go far from the place where it was turned back. Therefore the head's movement will not be straight even during a single sweep: it will make frequent short switchbacks (zigzags). This will trigger alarm, and the start of a recovery procedure if for example a turn-back is detected.

It is a significant challenge that the recovery procedure itself can be interrupted (or started) by a burst.

2.2 Hierarchical construction

In order to build a machine that can resist faults occuring independently of each other with some small probability, we take the approach suggested in [5], and implemented in [3] and [4] for the case of one-dimensional cellular automata, with some ideas from the tiling application of [2]. We will build a ***hierarchy of simulations***: machine M_1 simulates machine M_2 which simulates machine M_3, and so on. For simplicity we assume all these machines have the same program, and all simulations have the same blocksize.

One cell of machine M_3 is simulated by one colony of machine M_2. Correspondingly, one cell of M_2 is simulated by one colony of machine M_1. So one cell of M_3 is simulated by Q^2 cells of M_1. Further, one step of machine M_3 is simulated by one work period of M_2 of, say, $O(Q^2)$ steps. One step of M_2 is simulated by one work period of M_1, so one step of M_3 is simulated by $O(Q^4)$ steps of M_1.

Per construction, machine M_1 can withstand bursts of faults whose size is $\leqslant \beta$ for some constant parameter β, that are separated by some $O(Q^2)$ fault-free steps. Machines M_2, M_3, ... have the same program, so it would be natural to expect that machine M_1 can withstand also some *additional*, larger bursts of size $\leqslant \beta Q$ if those are separated by at least $O(Q^4)$ steps.

But a new obstacle arises. On the first level, damage caused by a big burst spans several colonies. The repair mechanism of machine M_1 outlined in Section 2.1 above is too local to recover from such extensive damage. This cannot be allowed, since then the whole hierarchy would stop working. So we add a new mechanism to M_1 that, more modestly, will just try to restore a large enough portion of the tape, so it can go on with the simulation of M_2, even if all original information was lost. For this, M_1 may need to rewrite an area as large as a few colonies.

This will enable the low-level recovery procedure of machine M_2 to restore eventually a higher-level order.

All machines above M_1 in the hierarchy are "virtual": the only hardware in the construction is machine M_1.

A tricky issue is "forced self-simulation": while we are constructing machine M_1 we want to give it the feature that it will simulate a machine M_2 that works just like M_1. The "forced" feature means that this simulation should work without any written program (that could be corrupted).

This will be achieved by a construction similar to the proof of the Kleene's fixed-point theorem (also called recursion theorem). We first fix a (simple) pro-

gramming language to express the transition function of a Turing machine. We write an interpreter for it in this same language (just as compilers for the C language are sometimes written in C). The program of the transition function of M_2 (essentially the same as that of M_1) in this language, is a string that will be "hard-wired" into the transition function of M_1, so that M_1, at the start of each work period, can write it on a working track of the base colony. Then the work period will interpret it, applying it to the data found there, resulting in the simulation of M_2.

In this way, an infinite sequence of simulations arises, in order to withstand larger and larger but sparser and sparser bursts of faults.

Since the M_1 uses the universal interpreter, which in turns simulates the same program, it is natural to ask how machine M_1 simulates a given Turing machine G that does the actual useful computation? For this task, we set aside a separate track on each machine M_i, on which some arbitrary other Turing machine can be simulated. The higher the level of the machine M_k that performs this "side-simulation", the higher the reliability. Thus, only the simulations $M_k \to M_{k+1}$ are forced, without program (that is a hard-wired program): the side simulations can rely on written programs, since the firm structure in the hierarchy $M_1, M_2, \ldots$ will support them reliably.

2.3 From combinatorial to probabilistic noise

The construction we gave in the previous subsection was related to increasing bursts that are not frequent. In essence, that noise model is combinatorial. To deal with probabilistic noise combinatorially, we stratify the set of faulty times *Noise* as follows. For a series of parameters β_k, V_k, we first remove "isolated bursts" of type (β_1, V_1) of elements of this set. (The notion of "isolated bursts" of type (β, V) will be defined appropriately.) Then, we remove isolated bursts of type (β_2, V_2) from the remaining set, and so on. It will be shown that with the appropriate choice of parameters, with probability 1, eventually nothing is left over from the set *Noise*.

A composition of two reliable simulations is even more reliable. We will see that a sufficiently large hierarchy of such simulations resists probabilistic noise.

2.4 Difficulties

Let us spell-out some of the main problems that the paper deals with, and some general ways they will be solved or avoided.

- A large burst of M_1 can modify the order of entire colonies or create new ones with gaps between them.

  To overcome this problem conceptually, we introduce the notion of a ***generalized Turing machine*** allowing for non-adjacent cells. Each such machine has a parameter B called the ***cell body size***. The cell body size of a Turing machine in Section 1.2 would still remain 1.

- What to do when the head is in a middle of an empty area where no structure exists? To ensure reliable passage across such areas, we will try to keep everything filled with cells, even if these are not part of the main computation.

- Noise can create areas over which the predictability of the simulated machine is limited. In these islands the (on this level) invisible structure of the underlying simulation may be destroyed. These areas should not simply be considered blank, since blankness implies predictable behavior. They will be called ***damaged***. When the head is in or near damage then even in the absence of new faults, the predictability of the behavior of the (simulated) machine will be severely limited.

- Due to limited predictability over damage, the definition of the generalized Turing machine stipulates a certain "magical" erasure of damage in case the head stays long enough on it. (Of course, this property needs to be implemented in simulation, which is one of the main burdens of the actual construction.) Once damage is erased the area will be re-populated with new cells. Their content is not important, what matters is the restoration of predictability.

- If local repair fails, a special rule will be invoked that reorganizes a larger part of the tape (of the size of a few colonies instead of only a few cells). This is the mechanism enabling the "magical" restoration on the next level.

2.5 A shortcut solution

A fault-tolerant one-dimensional cellular automaton is constructed in [4]. If our Turing machine could just simulate such an automaton, it would become fault-tolerant. This can indeed almost be done provided that the size of the computation is known in advance. The cellular automaton can be made finite, and we could define a "kind of" Turing machine with a *circular tape* simulating it. But this solution requires input size-dependent hardware.

It seems difficult to define a fault-tolerant sweeping behavior on a regular Turing machine needed to simulate cellular automaton, without recreating an entire hierarchical construction—as we are doing here.

3 Notation

Most notational conventions given here are common; some other ones will also be useful.

Natural numbers and integers. By $\mathbb{Z}$ we denote the set of integers.

$$\mathbb{Z}_{>0} = \{x : x \in \mathbb{Z},\ x > 0\},$$
$$\mathbb{Z}_{\geqslant 0} = \mathbb{N} = \{x : x \in \mathbb{Z},\ x \geqslant 0\}.$$

Intervals. We use the standard notation for intervals:

$$[a, b] = \{x : a \leqslant x \leqslant b\}, \quad [a, b) = \{x : a \leqslant x < b\},$$
$$(a, b] = \{x : a < x \leqslant b\}, \quad (a, b) = \{x : a < x < b\}.$$

We will also write $[a, b)$ in place of $[a, b) \cap \mathbb{Z}$, whenever this leads to no confusion. Instead of $[x + a, x + b)$, sometimes we will write

$$x + [a, b).$$

Ordered pairs. Ordered pairs are also denoted by (a, b), but it will be clear from the context whether we are referring to an ordered pair or open interval.

Comparing the order of a number and an interval. For a given number x and interval I, we write

$$x \geqslant I$$

if for every $y \in I$, $x \geqslant y$.

Distance. The distance between two real numbers x and y is defined in a usual way:

$$d(x, y) = |x - y|.$$

The ***distance of a point*** x ***from interval*** I is

$$d(x, I) = \min_{y \in I} d(x, y).$$

Ball, neighborhood, ring, stripe. A ***ball of radius*** $r > 0$***, centered at*** x is

$$B(x, r) = \{y : d(x, y) \leqslant r\}.$$

An r***-neighborhood of interval*** I is

$$\{x : d(x, I) \leqslant r\}.$$

An r***-ring*** around interval I is

$$\{x : d(x, I) \leqslant r \text{ and } x \notin I\}.$$

A r***-stripe to the right of interval*** I is

$$\{x : d(x, I) \leqslant r \text{ and } x \notin I \text{ and } x > I\}.$$

Logarithms. Unless specified differently, the base of logarithms throughout this work is 2.

4 Desribing a Turing machine

Let us introduce the tools allowing to describe the reliable Turing machine.

4.1 Universal Turing machine

We will describe our construction in terms of universal Turing machines, operating on binary strings as inputs and outputs. We define universal Turing machines in a way that allows for rather general "programs".

Definition 4.1 (Standard pairing)**.** For a (possibly empty) binary string $x = x(1)\cdots x(n)$ let us introduce the map

$$\langle x\rangle = 0^{|x|}1x,$$

Now we encode pairs, triples, and so on, of binary strings as follows:

$$\langle s, t\rangle = \langle s\rangle t,$$
$$\langle s, t, u\rangle = \langle\langle s, t\rangle, u\rangle,$$

and so on.

From now on, we will assume that our alphabets Σ, Γ are of the form $\Sigma = \{0,1\}^s$, $\Gamma = \{0,1\}^g$, that is our tape symbols and machine states are viewed as binary strings of a certain length. Also, if we write $\langle i, u\rangle$ where i is some number, it is understood that the number i is represented in a standard way by a binary string. ⌟

Definition 4.2 (Computation result, universal machine)**.** Assume that a Turing machine M starting on binary x, at some time t arrives at the first time at some final state. Then we look at the longest (possibly empty) binary string to be found starting at position 0 on the tape, and call it the ***computation result*** $M(x)$. We will write

$$M(x,y) = M(\langle x,y\rangle), \quad M(x,y,z) = M(\langle x,y,z\rangle),$$

and so on.

A Turing machine U is called ***universal*** among Turing machines with binary inputs and outputs, if for every Turing machine M, there is a binary string p_M such that for all x we have $U(p_M, x) = M(x)$. (This equality also means that the computation denoted on the left-hand side reaches a final state if and only if the computation on the right-hand side does.) ⌟

Let us introduce a special kind of universal Turing machines, to be used in expressing the transition functions of other Turing machines. These are just the Turing machines for which the so-called s_{mn} theorem of recursion theory holds with $s(x,y) = \langle x,y\rangle$.

Definition 4.3 (Flexible universal Turing machine)**.** A universal Turing machine will be called ***flexible*** if whenever p has the form $p = \langle p', p''\rangle$ then

$$U(p,x) = U(p', \langle p'', x\rangle).$$

Even if x has the form $x = \langle x', x''\rangle$, this definition chooses $U(p', \langle p'', x\rangle)$ over $U(\langle p, x'\rangle, x'')$, that is starts with parsing the first argument (this process converges, since x is shorter than $\langle x,y\rangle$). ⌟

It is easy to see that there are flexible universal Turing machines. On input $\langle p, x\rangle$, a flexible machine first checks whether its "program" p has the form $p = \langle p', p''\rangle$. If yes, then it applies p' to the pair $\langle p'', x\rangle$. (Otherwise it just applies p to x.)

Definition 4.4 (Transition program)**.** Consider an arbitrary Turing machine M with state set Γ, alphabet Σ, and transition function δ. A binary string π will be called a ***transition program*** of M if whenever $\delta(a, q) = (a', q', j)$ we have

$$U(\pi, a, q) = \langle a', q', j \rangle.$$

We will also require that the computation induced by the program makes $O(|p| + |a| + |q|)$ left-right turns, over a length tape $O(|p| + |a| + |q|)$. ⌟

The transition program just provides a way to compute the (local) transition function of M by the universal machine, it does not organize the rest of the simulation.

Remark 4.5. In the construction of universal Turing machines provided by the textbooks (though not in the original one given by Turing), the program is generally a string encoding a table for the transition function δ of the simulated machine M. Other types of program are imaginable: some simple transition functions can have much simpler programs. However, our fixed machine is good enough (similarly to the optimal machine for Kolmogorov complexity). If some machine U' simulates M via a very simple program q, then

$$M(x) = U'(q, x) = U(p_{U'}, \langle q, x \rangle) = U(\langle p_{U'}, q \rangle, x),$$

so U simulates this computation via the program $\langle p_{U'}, q \rangle$. ⌟

4.2 Rule language

In what follows we will describe the generalized Turing machines M_k for all k. They are all similar, differing only in the parameter k; the most important activity of M_k is to simulate M_{k+1}. The description will be uniform, except for the parameter k. We will denote therefore M_k simply by M, and M_{k+1} by M^*. Similarly we will denote the block size Q_k of the block code of the simulation simply by Q.

Instead of writing a huge table describing the transition function $\delta_k = \delta_,$, we present the transition function as a set of ***rules***. It will be then possible to write one *interpreter* program that carries out these rules; that program can be written for some fixed flexible universal machine Univ.

Each rule consists of some (nested) conditional statements, similar to the ones seen in an ordinary program: "**if** *condition* **then** *instruction* **else** *instruction*", where the condition is testing values of some fields of the state and the

observed cell, and the instruction can either be elementary, or itself a conditional statement. The elementary instructions are an ***assignment*** of a value to a field of the state or cell symbol, or a command to move the head. Rules can call other rules, but these calls will never form a cycle. Calling other rules is just a shorthand for nested conditions.

Even though rules are written like procedures of a program, they describe a single transition. When several consecutive statements are given, then they change different fields of the state or cell symbol, so they can be executed simultaneously.

Assignment of value x to a field y of the state or cell symbol will be denoted by $y \leftarrow x$. We will also use some conventions introduced by the C language: namely, $x \leftarrow x + 1$ and $x \leftarrow x - 1$ are abbreviated to $x{+}{+}$ and $x{-}{-}$ respectively.

Rules can also have parameters, like $\texttt{Swing}(a, b, u, v)$. Since each rule is called only a constant number of times in the whole program, the parametrized rule can be simply seen as a shorthand.

Mostly we will describe the rules using plain English, but it should always be clear that they are translatable into such rules.

For the machine M we are constructing, each state will be a tuple $q = (q_1, q_2, \ldots, q_k)$, where the individual elements of the tuple will be called ***fields***, and will have symbolic names. For example, we will have fields *Addr* and *Drift*, and may write q_1 as $q.Addr$ or just *Addr*, q_2 as $q.Drift$ or *Drift*, and so on.

Similarly for tape symbols. In order to distinguish fields of tape symbols from fields of the state, we will always start the name of a field of the tape symbols by the letter c. We have seen already one example of this, the field *cDir* of tape symbols in the definition of a generalized Turing machine.

In what follows we describe some of the most important fields we will use; others will be introduced later.

A properly formatted configuration of M splits the tape into blocks of Q consecutive cells called ***colonies***. One colony of the tape of the simulating machine represents one cell of the simulated machine. The colony that corresponds to the cell that the simulated machine is scanning is called the ***base colony*** (a precise definition will be based on the actual history of the work of M). Once the drift is known, the union of the base colony with the neighbor colony in the direction of the drift is called the ***extended base colony*** (this notion will need to be defined more carefully later).

A field of the state will be called the ***mode***. in the ***normal*** mode, the the machine is engaged in the regular business of simulation. The ***recovery*** mode tries to correct some local fault due to a couple of neighboring bursts, while the

healing, or ***rebuilding*** mode attempts restoring the colony structure on the scale of a couple of colonies:

$$Mode \in \{\text{Normal}, \text{Recovering}, \text{Healing}\}.$$

The content of each cell of the tape of M also has several fields. Some of these have names identical to fields of the state. In describing the transition rule of M we will write, for example, $q.Addr$ simply as $Addr$, and for the corresponding field of the observed cell symbol a we will write $a.cInfo$, or just $cInfo$. The array of values of the same field of the cells will be called a ***track***. Thus, we will talk about the *cHold* track of the tape, corresponding to the *cHold* field of cells.

Each field of a cell has also a possible value $\emptyset$ whose approximate meaning is "undefined".

Some fields and parameters are important enough to introduce them right away. The

$$cInfo, cState$$

track of a colony of M contain the strings that encode the content of the simulated cell of M^* and its simulated state respectively.

$$cProg$$

track stores the program of M^*, in an appropriate form to be interpreted by the simulation. The field

$$cAddr$$

of the cell shows the position of the cell in its colony: it takes values in $[0, Q)$. There is a corresponding $Addr$ field of the state. The

$$Sw$$

field counts the sweeps that the head makes during the work period. There is a corresponding cSw field in the cell. The direction $d \in \{-1, 0, 1\}$ in which the simulated head moves will be denoted by

$$Drift.$$

There is a corresponding field *cDrift*. The number of the last sweep of the work period will depend on the drift d, and will be denoted by

$$\text{Last}(d). \tag{1}$$

Cells will be designated as belonging to a number of possible ***kinds***, signaled by the field

$$cKind$$

with values

$$\text{Member, Target, Bridge, Stem, Vac.}$$

Here is a description of the role of these cell kinds. Normally, cells will have the kind Member. During the simulation, however, the elements of the colony that is to become the next base colony, will be made to have the kind Target. If the neighbor colony is not adjacent, then the cells in the structure during simulation that connects the base colony to it will be made to have kind Bridge. When the simulation kills a represented cell, then the constituent members of its colony will not be killed, they will get the kind Stem. The reason is that even the "vacuum" taking the place of the colony needs some structure allowing the head to traverse it later in a reliable way.

It is useful to group some of the fields especially important for the simulation structure into a "super" field,

$$Core = (Addr, Sw, Drift, Kind), cCore = (cAddr, cSw, cDrift, cKind). \quad (2)$$

The fields used in recovery mode are all collected as subfields of the field *Rec* of the state, and the field *cRec* of the cell state. They will be introduced in the definition of the recovery rule.

In particular, when the field

$$cRec.Core \quad (3)$$

is not 0, we will call the cell ***marked for recovery***.

Similarly, the fields used in healing mode are subfields of the field *Arb* of the state, and the field *cArb* of the cell state. In particular, when the field

$$cArb.Core \quad (4)$$

is not 0, we will call the cell ***marked for rebuild***.

5 Exploiting structure in the noise

5.1 Sparsity

Let us introduce a technique connecting the combinatorial and probabilistic noise models.

Definition 5.1 (Centered rectangles, isolation)**.** Let $\boldsymbol{r} = (r_1, r_2)$, $r_1, r_2 \geqslant 0$, be a two-dimensional nonnegative vector. An ***rectangle*** of radius $\boldsymbol{r}$ ***centered*** at $\boldsymbol{x}$ is

$$B(\boldsymbol{x}, \boldsymbol{r}) = \{\boldsymbol{y} : |y_i - x_i| \leqslant r_i, i = 1, 2\}. \tag{5}$$

Let $E \subseteq \mathbb{Z}^2$ be a two-dimensional set. A point $\boldsymbol{x}$ of E is $(\boldsymbol{r}, \boldsymbol{r}^*)$***-isolated*** if

$$E \cap B(\boldsymbol{x}, \boldsymbol{r}^*) \subseteq B(\boldsymbol{x}, \boldsymbol{r}).$$

Let

$$D(E, \boldsymbol{r}, \boldsymbol{r}^*) = \{\boldsymbol{x} \in E : \boldsymbol{x} \text{ is not } (\boldsymbol{r}, \boldsymbol{r}^*)\text{-isolated from } E\}. \tag{6}$$

⌟

Definition 5.2 (Sparsity)**.** Let $\beta \geqslant 9$ be a constant, and let $0 < B_1 < B_2 < \cdots$, $T_1 < T_2 < \cdots$ be sequences of positive integers to be fixed later.

For a two-dimensional set E, let $E^{(1)} = E$. For $k > 1$ we define recursively:

$$E^{(k+1)} = D(E^{(k)}, \beta(B_k, T_k), (B_{k+1}, T_{k+1})). \tag{7}$$

Set $E^{(k)}$ is called the k***-th residue*** of E.

Set E is $(\boldsymbol{r}, \boldsymbol{r}^*)$***-sparse*** if $D(E, \boldsymbol{r}, \boldsymbol{r}^*) = \emptyset$, that is it consists of $(\boldsymbol{r}, \boldsymbol{r}^*)$-isolated points. It is k***-sparse*** if $E^{(k+1)} = \emptyset$. It is simply ***sparse*** if $\bigcap_k E^{(k)} = \emptyset$. ⌟

The following lemma connects the above defined sparsity notions to the requirement of small fault probability.

Lemma 5.3 (Sparsity)**.** *Let $1 = B_1 \leqslant B_2 \leqslant \cdots$ and $1 = T_1 < T_2 < \cdots$ be sequences of integers with $Q_k = B_{k+1}/B_k$, $U_k = T_{k+1}/T_k$, and*

$$\lim_{k \to \infty} \frac{\log(U_k Q_k)}{1.5^k} = 0, \tag{8}$$

For sufficiently small ε, for every $k \geqslant 1$ the following holds. Let $E \subseteq \mathbb{Z} \times \mathbb{Z}_{\geqslant 0}$ be a random set with the property that each pair (p, t) belongs to E independently from the other ones with probability $\leqslant \varepsilon$.

Then for each point $\boldsymbol{x}$ and each k,

$$\mathbb{P}\{B(\boldsymbol{x}, (B_k, T_k)) \cap E^{(k)} \neq \emptyset\} < 2\varepsilon \cdot 2^{-1.5^k}.$$

As a consequence, the set E is sparse with probability 1.

Proof. Let $k = 1$. Rectangle $B(\boldsymbol{x}, (B_1, T_1))$ is a single point, hence the probability of our event is $< \varepsilon$. Let us prove the inequality by induction, for $k + 1$.

Note that our event depends at most on the rectangle $B(\boldsymbol{x}, 3(B_k, T_k))$. Let

$$p_k = 2\varepsilon \cdot 2^{-1.5^k}.$$

Suppose $\boldsymbol{y} \in E^{(k)} \cap B(\boldsymbol{x}, (B_{k+1}, T_{k+1}))$. Then, according to the definition of $E^{(k)}$, there is a point

$$\boldsymbol{z} \in B(\boldsymbol{y}, T_{k+1}) \cap E^{(k)} \setminus B(\boldsymbol{y}, \beta(B_k, T_k)). \tag{9}$$

Consider a standard partition of the (two-dimensional) space-time into rectangles $K_p = \boldsymbol{c}_p + [-B_k, B_k) \times [-T_k, T_k)$ with centers $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$. The rectangles K_i, K_j containing $\boldsymbol{y}$ and $\boldsymbol{z}$ respectively intersect $B(\boldsymbol{x}, 2(B_{k+1}, T_{k+1}))$. The triple-size rectangles $K_i' = c_i + [-3B_k, 3B_k) \times [-3T_k, 3T_k)$ and K_j' are disjoint, since (9) implies $|y_1 - z_1| > \beta B_k$ and $|y_2 - z_2| > \beta T_k$.

The set $E^{(k)}$ must intersect two rectangles K_i, K_j of size $2(B_k, T_k)$ separated by at least $4(B_k, T_k)$, of the big rectangle $B(\boldsymbol{x}, 2(B_{k+1}, T_{k+1}))$.

By the inductive hypothesis, the event $\mathcal{F}_i$ that K_i intersects E_k has probability bound p_k. It is independent of the event $\mathcal{F}_j$, since these events depend only on the triple size disjoint rectangles K_i' and K_j'.

The probability that both of these events hold is at most p_k^2. The number of possible rectangles K_p intersecting $B(\boldsymbol{x}, 2(B_{k+1}, T_{k+1}))$ is at most $C_k := ((2U_kQ_k) + 2)^2$, so the number of possible pairs of rectangles is at most $C_k^2/2$, bounding the probability of our event by

$$\begin{aligned} C_k^2 p_k^2/2 &= 2C_k^2 \varepsilon^2 2^{-1.5^{k+1}} \cdot 2^{-0.5 \cdot 1.5^k} \\ &= 2\varepsilon 2^{-1.5^{k+1}} \cdot \varepsilon C_k^2 2^{-0.5 \cdot 1.5^k}. \end{aligned}$$

Since $\lim_k \frac{\log (U_kQ_k)}{1.5^k} = 0$, the last factor is $\leqslant 1$ for sufficiently small ε. □

5.2 Error-correcting code

Let us add error-correcting features to block codes introduced in Definition 1.2.

Definition 5.4 (Error-correcting code)**.** A block code is (β, t)-***burst-error-correcting***, if for all $x \in \Sigma_2$, $y \in \Sigma_1^Q$ we have $\psi^*(y) = x$ whenever y differs from $\psi_*(x)$ in at most t intervals of size $\leqslant \beta$. ⌟

Example 5.5 (Repetition code)**.** Suppose that $Q \geqslant 3\beta$ is divisible by 3, $\Sigma_2 = \Sigma_1^{Q/3}$, $\psi_*(x) = xxx$. Let $\psi^*(y)$ be obtained as follows. If $y = y(1) \dots y(Q)$, then $x = \psi^*(y)$ is defined as follows: $x(i) = \mathrm{maj}(y(i), y(i + Q/3), y + 2Q/3)$. For all $\beta \leqslant Q/3$, this is a $(\beta, 1)$-burst-error-correcting code.

If we repeat 5 times instead of 3, we get a $(\beta, 2)$-burst-error-correcting code. Let us note that there are much more efficient such codes than just repetition. ⌟

Let us assume that a generalized Turing machine

$$M = (\Gamma, \Sigma, \delta, NonAdj, cDir, q_{\mathrm{start}}, F, B, T)$$

is used to simulate a generalized Turing machine

$$M^* = (\Gamma^*, \Sigma^*, \delta^*, NonAdj^*, cDir^*, q^*_{\mathrm{start}}, F^*, B^*, T^*).$$

We will assume that $\Gamma^* \cup \{\emptyset\}$, and the alphabet Σ^* are subsets of the set of binary strings $\{0,1\}^\ell$ for some $\ell < Q$ (we can always ignore some states or tape symbols, if we want).

We will store the coded information at distance

$$PadLen \tag{10}$$

from the end of our colony of size Q. So let (υ_*, υ^*) be a $(\beta, 2)$-burst-error-correcting block code

$$\upsilon_* : \{0,1\}^\ell \cup \{\emptyset\} \to \{0,1\}^{(Q-2\cdot PadLen)B}.$$

We could use, for example, the repetition code of Example 5.5. Other codes are also appropriate, but we require that they have some fixed programs p_{encode}, p_{decode} on the universal machine Univ, in the following sense:

$$\upsilon_*(x) = \mathrm{Univ}(p_{\mathrm{encode}}, x), \quad \upsilon^*(y) = \mathrm{Univ}(p_{\mathrm{decode}}, y).$$

Also, these programs must work in quadratic time and linear space on a one-tape Turing machine (as the repetition code certainly does).

Let us now define the block code (ψ_*, ψ^*) used in the definition of the configuration code (ϕ_*, ϕ^*) as outlined in Section 6.2. We define

$$\psi_*(a) = 0^{PadLen} \upsilon_*(a) 0^{PadLen}. \tag{11}$$

The decoded value $\psi^*(x)$ is obtained by first removing *PadLen* symbols from both ends of x to get x', and then computing $\upsilon^*(x')$.

To compute the configuration code ϕ_* from ψ_* we proceed first as done in Section 6.2, and then add the following initializations: In cells of the base colony and its left neighbor colony, the *cSw* and *cDrift* fields are set to Last(+1) − 1, 1, and Last(+1), 1 respectively. In the right neighbor colony, these values are Last(−1) and −1 respectively. In all other cells, these values are empty. The *cAddr* fields of each colony are filled properly: the *cAddr* of the jth cell of a colony is j mod B^*.

Decoding of configurations will be defined later, when defining the map Φ^* of the simulation. It is more complex since the location of the base colony must be found also in configurations that are not in the range of the encoding function, but arise as a result of a lot of noise.

6 The model

6.1 Generalized Turing machine

Standard Turing machines do not have operations like "creation" or "killing" of cells, nor do they allow for cells to be non-adjacent. We introduce here a ***generalized Turing machine***. It depends on an integer $B \geqslant 1$ that denotes the cell body size, and an upper bound T on the transition time. These parameters are convenient since they provide the illusion that the different Turing machines in the hierarchy of simulations all operate on the same linear space. Even if the notions of cells, alphabet and state are different for each machine of the hierarchy, at least the notion of a *location on the tape* is the same.

Definition 6.1 (Generalized Turing machine)**.** A ***generalized Turing machine*** M is defined by a tuple

$$(\Gamma, \Sigma, \delta, NonAdj, cDir, q_{\text{start}}, F, B, T), \tag{12}$$

where Γ and Σ are finite sets called the ***set of states*** and the ***alphabet*** respectively,

$$\delta : \Sigma \times \Gamma \to \Sigma \times \Gamma \times \{-1, 0, 1\},$$

is the ***transition function*** The function ("field") $NonAdj\colon \Gamma \to \{\mathsf{true}, \mathsf{false}\}$ of the state will show whether the last move was from a non-adjacent cell. The function ("field") $cDir\colon \Sigma \to \{-1, 0, 1\}$ of the cell content needs to always point towards the head, so the transition function δ is required to have the property that if $(a', q', j) = \delta(a, q)$ then $a'.cDir = j$.

The role of starting state q_{start} and final states in F is as before. The integer $B \geqslant 1$ is called the ***cell body size***, and the real number T is a bound on the transition time.

Among the elements of the tape alphabet Σ, we distinguish the elements $0, 1, Bad$, Vac. The role of the symbols Bad and Vac will be clarified below. ⌟

Definition 6.2 (Configuration)**.** Consider a generalized Turing machine (12). A ***configuration*** is a tuple

$$(q, A, h, \tilde{h}),$$

where $q \in \Gamma$, $h, \tilde{h} \in \mathbb{Z}$ and $A \in \Sigma^{\mathbb{Z}}$. As before, the array A is the tape configuration. The ***damage set*** is defined as

$$A.Damage = \{p : A[p] = Bad\}.$$

We say that there is a ***cell*** at position $p \in \mathbb{Z}$ if $A[p] \notin \{\text{Vac}, Bad\}$. In this case, we call the interval $p + [0, B)$ the ***body*** of this cell. Cells must be at distance $\geqslant B$ from each other, that is their bodies must not intersect. They are called ***adjacent*** if the distance is exactly B.

For all cells p, the value $A[p].cDir$ is required to point towards the head h, that is

$$A[p].cDir = \text{sign}(h - p).$$

Whenever $A[h] \neq Bad$ there must be a cell at position $\tilde{h}$ called the ***current cell*** with a body within $2.5B$ from h.

The array A is Vac everywhere but in finitely many positions.

Let

$$\text{Configs}_M$$

denote the set of all possible configurations of a Turing machine M. ⌟

It is natural to name a sequence of configurations that is conceivable as a computation (faulty or not) of a Turing machine as "history". The histories that obey the transition function then could be called "trajectories". In what follows we will stretch and generalize this notion to encompass also some limited violations of the transition function.

In connection with any underlying Turing machine with a given starting configuration, we will denote by

$$Noise \subseteq \mathbb{Z} \times \mathbb{Z}_{\geqslant 0} \tag{13}$$

the set of space-time points (p, t), such that a fault occurs at time t when the head is at position p.

Definition 6.3 (History)**.** Let us be given a generalized Turing machine (12). Consider a sequence $\eta = (\eta(0), \eta(1), \dots)$ of configurations with $\eta(t) = (q(t), A(t), h(t), \tilde{h}(t))$, along with a noise set *Noise*. The ***switching times*** of this sequence are the times when one of the following can change: the state, the position $\tilde{h}(t)$ of the current cell, or the state of the current cell. The interval between two consecutive switching times is the ***dwell period***. The pair

$$(\eta, Noise)$$

will be called a ***history*** of machine M if the following conditions hold.

- We have $|h(t) - h(t')| \leqslant |t' - t|$.
- In two consecutive configurations, the content $A(t)[p]$ of the positions not in $h(t) + [-B, B)$, remains the same: $A(t+1)[n] = A(t)[n]$ for all $n \notin h(t) + [-B, B)$.
- At each noise-free switching time the head is on the new current cell, that is $\tilde{h}(t) = h(t)$. In particular, when at a switching time a current cell becomes Vac, the head must already be on another (current) cell.
- The length of any dwell period in which the head does not intersect noise or damage, is at most T.

Let

$$\text{Histories}_M$$

denote the set of all possible histories of M.

We say that a cell ***dies*** in a history if it becomes Vac.

⌟

The transition function δ of a generalized Turing machine imposes constraints on histories: those histories obeying the constraints will be called trajectories.

Definition 6.4. Suppose that the machine is in a state q, with current cell x, with cell content a, let $(a', q', j) = \delta(a, q)$. We say that the new content of cell x (including when it dies), the direction of the new position y from x, and the new state are ***directed by the transition function*** if the following holds. The new content of x is a', and the direction of y from x is j. In the new state q we have $q.NonAdj = \mathsf{false}$ if $j = 0$ or the new current cell is adjacent, and true otherwise. ⌟

Definition 6.5 (Trajectory)**.** The definition of a trajectory depends on a constant

$$c_1 > 0$$

to be fixed later.

A history $(\eta, Noise)$ of a generalized Turing machine (12) with $\eta(t) = (q(t), A(t), h(t), \tilde{h}(t))$ is called a ***trajectory*** of M if the following conditions hold, during any noise-free time interval. Denote

$$(a, q, p) = (A(t)[\tilde{h}(t)], q(t), \tilde{h}(t)),$$
$$(a', q', j) = \delta(a, q).$$

Transition function. Suppose that there is a switch, and the shortest interval containing the body of the current cell x and the new cell y is at a distance at least $0.5B$ from damage.

Then the new state, the cell content of x (including when it dies), and the direction of y from x are directed by the transition function. If y did not exist before then it is adjacent to x.

Further the length of the dwell period is bounded by T.

Spill. The damage may spill only to at most a distance $2B$ beyond its place at time t.

Attacking damage from outside. Suppose that the body of the current cell x is disjoint from damage, and at least $0.5B$ removed from damage on the left. Suppose further that the transition function directs the head towards the right. Then whenever the head comes left of x again, the damage will recede by a distance at least B from the body of x.

A similar property is required when "left" and "right" are interchanged.

Clearing damage from within. Let I be a space interval of size $3B$. If the head spends a total time of at least $c_1 T$ inside I (while possibly entering repeatedly), then I becomes damage-free.

⌟

Until this moment, we used the term "simulation" to denote a correspondence between configurations of two machines which remains preserved during the computation.

Definition 6.6 (Simulation)**.** Let M_1, M_2 be two generalized Turing machines, and let

$$\varphi_* : \mathrm{Configs}_{M_2} \to \mathrm{Configs}_{M_1}$$

be a mapping from configurations of M_2 to those of M_1, such that it maps starting configurations into starting configurations. We will call such a map a ***configuration encoding***.

Let

$$\Phi^* : \mathrm{Histories}_{M_1} \to \mathrm{Histories}_{M_2}$$

be a mapping. The pair (ϕ_*, Φ^*) is called a ***simulation*** (of M_2 by M_1) if for every trajectory (η, Noise) with initial configuration $\eta(0) = \phi_*(\xi)$, the history $(\eta^*, \mathit{Noise}^*) = \Phi^*(\eta, \mathit{Noise})$ is a trajectory of machine M_2.

We say that M_1 ***simulates*** M_2 if there is a simulation (φ_*, Φ^*) of M_2 by M_1. ⌟

6.2 Hierarchical codes

Recall the notion of a code in Definition 1.2.

Definition 6.7 (Code on configurations)**.** Consider two generalized Turing macines M_1, M_2 with the corresponding state spaces, alphabets and transition functions, and an integer $Q \geqslant 1$. We require

$$B_2 = QB_1. \tag{14}$$

Assume that a block code

$$\psi_* : \Sigma_2 \times (\Gamma_2 \cup \{\emptyset\}) \to \Sigma_1^Q$$

is given, with an appropriate decoding function, ψ^*. With $(a, q) \in \Sigma_2 \times (\Gamma_2 \cup \{\emptyset\})$, symbol a is interpreted the content of some tape square. The value q is the state of M_2 provided the head is observing this square, and $\emptyset$ if it is not.

Block code (ψ_*, ψ^*) gives rise to a ***code on configurations***, that is a pair of functions

$$\phi_* : \text{Configs}_{M_2} \to \text{Configs}_{M_1}, \quad \phi^* : \text{Configs}_{M_1} \to \text{Configs}_{M_2}$$

that encodes configurations of M_2 into configurations of M_1.

Let ξ be a configuration of M_2. We set $\phi_*(\xi).\text{pos} = \xi.\text{pos}$, $\phi_*.\text{state}$ = the starting state of M_1,

$$\phi_*(\xi).\text{tape}[iB_2, \ldots, (i+1)B_2 - B_1] = \psi_*(\xi.\text{tape}[i], s)$$

where $s = \xi.\text{state}$ if $i = \xi.\text{pos}$, and $\emptyset$ otherwise, with a slight *modification*: $\phi_*(\xi).\text{tape}[i].cDir$, is set to point towards the head $\xi.\text{pos}$. Decoding is the inverse of this process (need not succeed on all possible configurations of M_1). ⌟

Not all configurations can be obtained by encoding. We distinguish those that can.

Definition 6.8 (Code configuration)**.** A configuration ξ is called a ***code configuration*** if it has the form $\xi = \phi_*(\zeta)$. ⌟

Definition 6.9 (Hierarchical code)**.** For $k \geqslant 1$, let Σ_k be an alphabet, Γ_k be a set of states of a generalized Turing machine M_k. Let $Q_k > 0$ be an integer colony size, let ϕ_k be a code on configurations defined by a block code

$$\psi_k : \Sigma_{k+1} \times (\Gamma_{k+1} \cup \{\emptyset\}) \to \Sigma_k^{Q_k}$$

as in Definition 6.7. The sequence of triples $(\Sigma_k, \Gamma_k, \phi_k)$, $(k \geqslant 1)$, is called a ***hierarchical code***. For the given hierarchical code, the configuration ξ^1 of M_1 is called a ***hierarchical code configuration*** if a sequence of configurations $\xi^2, \xi^3, \ldots$ of $M_2, M_3, \ldots$ exists with

$$\xi^k = \phi_{*k}(\xi^{k+1})$$

for all k. (Of course, then whole sequence is determined by ξ^1.) ⌟

Let us give a name to the object that we want to construct.

Definition 6.10 (A tower)**.** Let $M_1, M_2, \ldots$, be a sequence of generalized Turing machines, let $\phi_1, \phi_2, \ldots$ be a hierarchical code for this sequence, let ξ^1 be a hierarchical code configuration for it, where ξ^k is an initial configuration of M_k for each k. Let further be a sequence of mappings $\Phi_1, \Phi_2, \ldots$ be given such that for each k, the pair (ϕ_{k*}, Φ_k^*), is a simulation of M_{k+1} by M_k. Such an object is called a ***tower***. ⌟

The main task of the work will be the definition of a tower, since the simulation property is highly nontrivial.

7 Simulation structure

The first task of each simulation step is to check if the state of the machine and the current cell are in a certain relationship. When this condition is breached (due to some burst), machine calls `Alarm` — a rule that will initiate the recovery. Below, we will specify these conditions, and aggregate them into one, that we call the ***coordination***.

The sketch of the main rule of machine M is given in Rule 7.1, where the `Compute` and `Transfer` rules will be outlined below.

Rule 7.1: Main rule

if the mode is normal **then**
  if not Coordinated **then** `Alarm`
  else if $1 \leqslant Sw <$ TransferSw(1) **then** `Compute`
  else if TransferSw(1) $\leqslant Sw <$ Last **then** `Transfer`
  else if Last $\leqslant Sw$ **then** move the head to the new base.

7.1 Sweep counter and direction

The global sweeping movement of the head will be controlled by the parametrized rule

$$\texttt{Swing}(a, b, u, v).$$

This rule makes the head swing between two extreme points a, b, while the counter Sw increases from value u to value v. The Sw value is incremented at the "turns" a, b (and is also recorded on the track cSw).

The sweep direction δ of the simulating head is derived from Sw, $Addr$ and the current value Dir in the following way. On arrival of the head to an endpoint (that is when $Dir \neq 0$ and $Addr \in \{a, b\}$), the values Sw and cSw are incremented and Dir is set to 0. In all other cases, the sweep direction is determined using the formula

$$\mathrm{dir}(s) = (-1)^{s+1} \tag{15}$$

as follows:

$$\delta = \begin{cases} 0 & \text{if } Addr \in \{a, b\} \text{ and } Dir \neq 0, \\ \mathrm{dir}(Sw) & \text{otherwise.} \end{cases} \tag{16}$$

As we mentioned, each sweep will be broken up into zigzags to allow the detection of premature turn-back. At each non-zigging step, $Addr \leftarrow Addr + \delta$.

As an example of rules, we present the the zigging rule. It uses the following parameters:

Definition 7.1. Let $Z = \beta^2$, $Z_b = \beta$. ⌟

Certain bursts may turn back the head prematurely, causing a skip in the simulation. We want to prevent this, since we would like the size of the structural repairs to be comparable to the burst size. Zigging is introduced for this purpose. Rule 7.2, itself uses the rule `Move`(d), which we will defined later. A characteristic of this rule is that it changes nothing on the tape: all its control information is in the state. The rule repeats the following cycle: first it moves forward $Z - Z_b$, moving ahead the ***front***, and performing all necessary operations of the computation. Then it moves backward and forward by Z steps, not changing anything on the tape (but as we will see, some consistencies will be checked). The field $ZigDepth$ of the state measures the distance from the front during the switchback. From now on, we will assume that Q is a multiple of $Z - Z_b$.

7.2 Computation phase

As shown in Rule 7.1 descibing a top-down view of the simulation, the first phase of the simulation computes new values for state of the simulated machine

Rule 7.2: `Zigzag`(d)

// $d \in \{-1, 1\}$ is the direction of progress.
if $ZigDir = -1$ **and** $((ZigDepth = 0$ **and** $(Z - 4\beta)|Addr)$
or $0 < ZigDepth < Z)$ **then**
    $ZigDepth{+}{+}$
    if $ZigDepth = Z - 1$ **then** $ZigDir = 1$
    `Move`$(-d)$
else if $ZigDir = 1$ **or** $(ZigDepth = 0$ **and** $(Z - 4\beta) \not| Addr)$ **then**
    if $ZigDepth > 0$ **then** $ZigDepth{-}{-}$ **else** $ZigDir \leftarrow -1$
    `Move`(d)

M^* represented on track *cState*, the direction of the move of the head of M^* (represented in the *cDrift* field of each cell of the colony of M), and the simulated cell state of M^* represented on the track *cInfo*. During this rule, the head sweeps the base colony.

The rule `Compute` essentially repeats five times the following ***stages***: decoding, applying the transition, encoding, checking for destruction.

Due to the possibility of encountering a much larger burst of faults than this rule can handle, some extra rules will then be applied that we will specify later: `UsefulComp`.

In more detail:

1. In the first sweep of the work period, set $cAddr \leftarrow cAddr \bmod Q$ and $cKind \leftarrow$ Member.

2. For every $j = 1, \ldots, 5$, if $Addr \in \{0, \ldots, Q - 1\}$ do

    a) Calling by g the string found on the *cState* track of the base colony between addresses *PadLen* and $Q - PadLen$, decode it into string $\tilde{g} = \upsilon^*(g)$ (this should be the current state of the simulated machine), and store it on some auxiliary track in the base colony. Do this by simulating the universal machine on the *cProg* track, $\tilde{g} = \mathrm{Univ}(p_{\mathrm{decode}}, g)$. Proceed similarly with the string a found on the *cInfo* track of the base colony, between addresses *PadLen* and $Q - PadLen$, to get $\tilde{a} = \upsilon^*(a)$ (this should be the observed tape symbol of the simulated machine).

b) Compute the value $(a', g', d) = \delta^*(\tilde{a}, \tilde{g})$. Since the neither the table nor any program of the transition function δ^* is written explicitly anywhere, this "self-simulation" step needs some elaboration, see Section 7.3.

c) Write the encoded new state $\upsilon_*(g')$ onto the *cHold*[j].*State* track of the base colony between positions *PadLenB* and $Q - PadLenB$. Similarly, write the encoded new observed cell content $\upsilon_*(a')$ onto the *cHold*[j].*Info* track. Write d into the *cHold*[j].*Drift* field of *each cell* of the base colony.

Special action needs to be taken in case the new state g' is a vacant one, that is $g'.Kind^* = \text{Vac}^*$. In this case, write 1 onto the *cHold*[j].*Doomed* track (else 0).

3. Repeat the following twice (hoping that at least one repetition will be burst-free):

   Sweeping through the base colony, at each cell compute the majority of *cHold*[j].*Info*, $j = 1, \ldots, 5$, and write into the field *cInfo*. Proceed similarly, and simultaneously, with *State* and *Drift*.

4. Call the rule `UsefulComp` that we will specify later.

It can be arranged—and we assume so—that the total number of sweeps of this phase, and thus the starting sweep number of the next phase, depends only on Q.

7.3 Self-simulation

In describing the rule of the computation phase, in the step 2b of Section 7.2, we said that machine M writes the code p^* of M^* onto the *cProg* track, without saying how this is done. Here we give the details.

New primitives

We will make use of a special track

cWork

of the cells and the special field

Index

of the state of machine M that can store a certain address of a colony.

Recall from Section 4.2 that the program of our machine is a list of nested "**if** *condition* **then** *instruction* **else** *instruction*" statements. As such, it can be represented as a binary string

$$R.$$

If one writes out all details of the construction of the present paper, this string R becomes completely explicit, an absolute constant. But in the reasoning below, we treat it as a parameter.

There is a couple of ***extra primitives*** in the rules. First, they have access to the parameter k of machine $M = M_k$, to define the transition function

$$\delta_{R,k}(a, q).$$

The other, more important, new primitive is a special instruction

$$\texttt{WriteRulesBit}$$

in the rules. When called, this instruction makes the assignment $cWork \leftarrow R(Index)$. This is the key to self-simulation: *the program has access to its own bits*. If $Index = i$ then it writes $R(i)$ onto the current position of the *cWork* track.

Simulating the rules

By convention, in our fixed flexible universal machine Univ, program p and input x produce an output $\mathrm{Univ}(p, x)$. Since the structure of all rules is very simple, they can be read and interpreted by Univ in reasonable time:

Theorem 2. *There is a constant string called Interpr with the property that for all positive integers k, string R that is a sequence of rules, and bit strings* $a \in \Sigma_k$, $q \in \Gamma_k$*:*

$$\mathrm{Univ}(Interpr, R, 0^k, a, q) = \delta_{R,k}(a, q).$$

The computation on Univ *takes time* $O(|R| \cdot (|a| + |q|))$.

The proof parses and implements the rules in the string R; each of these rules checks and writes a constant number of fields.

Implementing the `WriteRulesBit` instruction is straightforward: Machine Univ determines the number i represented by the simulated *Index* field, looks up $R(i)$ in R, and writes it into the simulated *cWork* field.

Note that there is no circularity in these definitions:

- The instruction `WriteRulesBit` is written *literally* in R in the appropriate place, as "`WriteRulesBit`". The string R is *not part* of the rules (that is of itself).
- On the other hand, the computation in $\mathrm{Univ}(\mathit{Interpr}, R, 0^k, a, q)$ has *explicit* access to the string R as one of the inputs.

Let us show the computation step invoking the "self-simulation" in detail. In the earlier outline, step 2b of Section 7.2, said to compute $\delta^*(\tilde{a}, \tilde{g})$ (for the present discussion, we will just consider computing $\delta^*(a, q) = \delta_{k+1}(a, q)$), where $\delta = \delta_k$, and it is assumed that a and q are available on two appropriate auxiliary tracks. We give more detail now of how to implement this step:

1. Onto the *cWork* track, write the string R. To do this, for *Index* running from 1 to $|R|$, execute the instruction `WriteRulesBit` and move right. Now, on the *cWork* track, replace it with $\langle \mathit{Interpr}, 0^{k+1}, R, a, q\rangle$. Here, string *Interpr* is a constant, so it is just hardwired. String R already has been made available. String 0^{k+1} can be written since the parameter k is available. Strings a, q are available on the tracks where they were stored.
2. Simulate the universal automaton Univ on track *cWork*: it computes $\delta_{R,k+1}(a, q) = \mathrm{Univ}(\mathit{Interpr}, R, 0^{k+1}, a, q)$ as needed.

   This achieves the forced simulation. Note what we achieved:

   - On level 1, the transition function $\delta_{R,1}(a, q)$ is defined completely when the rule string R is given. It has the forced simulation property by definition, and string R is *"hard-wired"* into it in the following way. If $(a', q', d) = \delta_{R,1}(a, q)$, then

     $$a'.\mathit{cWork} = R(q.\mathit{Index})$$

     whenever $q.\mathit{Index}$ represents a number between 1 and $|R|$. and the values $q.\mathit{Sw}$, $q.\mathit{Addr}$ satisfy the conditions under which the instruction `WriteRulesBit` is called in the rules (written in R).
   - The forced simulation property of the *simulated* transition function $\delta_{R,k+1}(\cdot, \cdot)$ is achieved by the above defined computation step—which *relies on* the forced simulation property of $\delta_{R,k}(\cdot, \cdot)$.

Remark 7.2. This construction resembles the proof of Kleene's fixed-point theorem. ⌟

7.4 Transfer phase

If the simulated machine head moves left or right, then another phase will be added to the simulation: transferring the simulated state information to the neighbor colony, and to move there. Let us call the direction of the transfer the ***drift***.

During this phase, the range of the head includes the base colony and the neighbor colony determined by the drift, including a possible "bridge" between them.

The sweep in which we start transferring in direction δ is called TransferSw(δ), the ***transfer sweep***. We have TransferSw(-1) = TransferSw(1) + 1.

General structure of the phase

We will make use of some extra rules that we will specify in more detail later, but whose role is spelled out here.

The phase consists of the following actions.

1. Spread the value δ found in the cells of the *cDrift* track (they should all be the same) onto the neighbor colony in direction δ.

   There are some details to handle in case the neighbor colony is not adjacent: see Section 7.4.

2. For $i = 1, 2, 3$:

   Copy the content of *cState* track of the base colony to the $cHold[i].State$ track of the neighbor colony.

3. Repeat the following twice:

   Assign the field majority: $cState \leftarrow \mathrm{maj}(cHold[1 \ldots 3].State)$ in all cells of the neighbor colony.

4. If $Drift = 1$, then move right to the left endcell of the neighbor colony (else you are already there).

5. In the last sweep (possibly identical with the move step above), in the base colony, if the majority of $cHold[j].Doomed$, $j = 1, \ldots, 5$, is 1 then turn the scanned cell into a stem cell: in other words, carry out the destruction.

Transfer to a non-adjacent colony

Let us address the situation when the neighbor colony is not adjacent.

Definition 7.3 (Adjacency of cells)**.** Cells a and b are ***adjacent*** if $|a - b| = B$. Otherwise, if $B < |a - b| < 2B$, then a and b are two ***non-adjacent neighbor cells***. For the sake of the present discussion, a ***colony*** is a sequence of Q adjacent cells whose *cAddr* value runs from 0 to $Q - 1$.

If the bodies of two cells are not adjacent, but are at a distance $< B$ then the space between them is called a ***small gap***. We also call a small gap such a space between the bodies of two colonies. On the other hand, if the distance of the bodies of two colonies is $> B$ but $< QB$ then the space between them is called a ***large gap***. ⌟

In the transfer phase, in order to know in a robust local way where the head is, the *cKind* field of the cells visited will be set as follows. The base colony has cells of kind Member to begin with. The kind of the cells of the neighbor colony, the target of the transfer, will be set as Target for the duration of the transfer. However, in the first transfer sweep, the cells adjacent to the base colony get the kind Bridge. A bridge can be extended over an opposite old bridge ("old" meaning that its *Sw* is maximal) or into an empty area, killing opposite bridge cells or stem cells in the process. The bridge has its own addresses, starting from 0 if it extends to the right, and from $Q - 1$ if it extends to the left. If while forming a bridge, another colony is encountered before the bridge grows to length QB), then this new colony's cells will get the kind Target, and add it all to the workspace. There can be a gap of size $< B$ between the bridge and the neighbor colony. Otherwise the bridge itself becomes this neighbor colony, and its cell kinds are turned to Target on the way back.

Recall that the *NonAdj* field of the state determines if the current cell is not adjacent to the cell where the head came from. After the transfer stage, we update the $NonAdj^*$ field encoded in the *cState* track of the neighbor colony: it becomes 1 if either there is a nonempty bridge, or there is a gap (found with the help of the *NonAdj* field) between the base colony and the neighbor colony.

This is done in part 2 of Section 7.4 again three times, storing candidate values into $cHold[j].NonAdj$ and repeated with everything else.

7.5 Notes on zigging

The zigging rule was introduced in Section 7.1. We add now the following refinements to it.

When the head steps outside of the extended base colony and does not find a cell, then it creates a stem cell whose *cDrift* is set to $-Dir$, where *Dir* is the field

of the state storing the direction of the last step (see 4.2). While zigging outside the extended base colony in normal mode, we allow one small gap, next to the extended base colony (other situations cause alarm).

8 Integrity of the structure

The main part of the simulation uses an error-correcting code to protect information stored in *Info* and *State* fields. However, faults can ruin the simulation structure and disrupt the simulation itself. The error-correcting capabilities of the code used to store the information on the *Info* and *State* tracks, will preserve the content of these tracks as long as coding-decoding process implemented in the simulation is carried out. The structural integrity of a configuration is maintained with the help of a small number of fields. Below we outline the necessary relations among them allowing the identification and correction of local damage.

8.1 Healthy configuration

A configuration with local structural integrity will be called healthy. No cell in such a configuration should have marks of a recovery or rebuild procedure. Larger bursts introduce new, non-local anomalies: these we will "legalize", since they might encode some conceivable configuration of the machine we simulate. Cells of a healthy configuration are grouped into gapless colonies, and a few transitional segments described below.

Definition 8.1 (Segments)**.** A ***(homogenous) segment*** is a sequence of adjacent neighbor cells of the same kind, with addresses growing to the right, with the same value of *Sw* and *cDrift*, or a sequence of neighboring (not necessarily adjacent) stem cells (this will be called a ***desert***). Its ***left end*** is the left edge of its first cell, and its ***right end*** is the right edge of its last cell.

It is a ***colony*** if the addresses grow from 0 to $Q-1$. It is a ***bridge*** if it consists of bridge cells, a ***target*** if it consists of target cells.

A boundary is called ***rigid*** if its address is the end address of a colony in the same direction.

A ***boundary pair*** is a right boundary followed by a left boundary that is at distance $< B$ from it. It is a ***hole*** if the distance is greater than 0. It is ***rigid*** if at least one of its elements is. ⌟

In a healthy configuration, cells fall into certain categories. Outer cells are member cells in colonies other than the ones that are currently being manipulated.

Definition 8.2 (Outer cells)**.** Recall the definition of the sweep value Last(δ) from (1). For $\delta \in \{-1, 1\}$, if a cell is stem or has $0 \leqslant cAddr < Q$, $cDrift = \delta$, $cSw = \text{Last}(\delta)$ then it will be called a ***right outer cell*** if $\delta = -1$. It is a ***left outer cell*** if $\delta = 1$. ⌟

According to this definition, a stem cell is both left and right outer cell.

Recall the definition of the ***transfer sweep*** TransferSw(δ) in Section 7.4, if $\delta \neq 0$. (There is no transfer sweep if $\delta = 0$.)

Definition 8.3 (Healthy configuration)**.** A configuration ξ of a generalized Turing machine M, is ***healthy***, if the mode is normal, and the following conditions hold. Let $\delta = Drift$.

Normality. No cell in I is marked for recovery or healing. That is, for every $x \in I$, $cRec.Core = 0$ and $cArb.Core = 0$ (see the definition of marking after (3) and after (4), respectively).

Segments. All cells can be grouped into the following:

- A segment of the ***base colony*** consisting of member cells.
- ***Outer colonies***, consisting of outer member cells.
- A possible bridge of size $< Q$.
- A segment of a possible target, defined by the value of cSw in its cells.
- Desert filling out the gaps between these other parts.

The ***base*** is defined by counting back from $Addr$ such that, if the head passes from a target cell to a bridge or member, or from a bridge to a member cell, we set $Addr$ to $cAddr$. The cell closer to the base where the $Addr$ is adjusted during this count-back is called the ***address jump***.

The colony starting at the base is the base colony, and it consists of member cells. To describe the other segments, we consider several cases.

- If $\delta = 0$ or $Sw < \text{TransferSw}(\delta)$, then there are no bridge or target cells.
- If $\text{TransferSw}(\delta) + 1 < Sw <$ (the last two sweeps for δ), then there is a target colony in the direction δ. If its distance from the base colony is $\geqslant B$ then there is a bridge adjacent to the base colony filling the gap between the base colony and the target colony. We define the ***extended base colony*** by extending the base colony by the target colony, with the possible bridge in between.

- If $Sw = \text{TransferSw}(\delta)$ then the above described situation is in the making, as a bridge segment is being built up in direction δ, or after that, a target segment is being built in direction δ, converting member cells into target cells.
- If $Sw = \text{TransferSw}(\delta) + 1$ and there is still not a complete target colony, then this can only happen if the whole bridge is being converted into a target, as the head travels in direction $-\delta$.
- In the last sweep, the target cells are being converted into member cells.

These are the only possible segments to be seen in a healthy area.

The front. The farthest position $\text{front}(\xi)$ to which the head has advanced before starting a new backwards zig is called the ***front***.

Workspace. Suppose that $\text{front}(\xi)$ is inside the extended base colony. The ***workspace*** is an interval of non-outer cells, such that:

- For $Sw < \text{TransferSw}(1)$, it is equal to the base colony.
- In case of $Sw = \text{TransferSw}(1)$, it is the smallest interval including the base colony and the cell neighboring to $\text{front}(\xi)$ on the side of the base colony.
- In case of $Sw = \text{TransferSw}(-1)$, it is the smallest interval including the base colony, the right neighbor colony, and the cell neighboring to $\text{front}(\xi)$ on the side of the base colony.
- If $\text{TransferSw}(-1) < Sw < \text{Last}(\delta)$, then it is equal to the extended base colony.
- When $Sw = \text{Last}(\delta)$, it is the smallest interval including the future base colony and $\text{front}(\xi)$.

《Peter: I hope to avoid yarding.》

Drift. If $Sw \geqslant \text{TransferSw}(\delta)$ or $Sw = 1$ then *cDrift* is constant on the workspace.

Coordination. In a healthy configuration, each $Core = (Addr, Sw, Drift, Kind)$ value along with Z determines uniquely the *cCore* value of the cell it is coordinated with, with the following exceptions.

- During the first transferring sweep, while creating a bridge between the base colony and the target colony, the front can be a stem cell or the first cell of an outer colony.
- Every jump backward from the target colony can end up on the last cell of a bridge (whose address is not recorded in the state) or the last cell of the base colony.

To compute the values in question, calculate Z steps backwards from the front, referring to the properties listed above.

⌟

Based on the above, let us classify the boundary pairs possible in a healthy configuration. Rigid pairs:

(r1) Between a base colony and its extension bridge.

(r2) Between an outer colony (possibly extended by an old bridge) or a desert, and the workspace: base colony or target.

(r3) Between a bridge and the target or an outer colony.

Non-rigid pairs:

(nr1) Between sweep values differing by 1, between a new bridge and the target that it is possibly being converted into, between a target in direction δ and the remaining segment of member cells, in direction $-\delta$, or between the target and the member cells replacing it in the last sweep.

(nr2) Boundary of a new bridge in direction δ not within distance B of a rigid boundary of a colony or target in direction $-\delta$. On the other side is either desert or the boundary of an old bridge.

The following is worth noting.

Lemma 8.4. *In a healthy configuration, any interval of size $< Q$ contains at most 3 boundary pairs, only one of which can be a hole.*

Proof. Any interval of size $< Q$ contains at most one rigid left boundary and one rigid right boundary. Of the boundary pairs containing the other types, only one can be present: indeed, they all coincide with the location of the front. □

Definition 8.5. A configuration is ***super healthy*** if in addition to the requirements of health, in each colony, whenever the head is not in the last sweep, the *Info* and *State* tracks contain valid codewords as defined in Section 5.2.

A configuration ξ defined on an interval I is ***(super) healthy*** on I if it can be extended to a (super) healthy configuration. ⌟

Note that health only depends on the fields in *Core* and the zigging field Z in the state, further the *cCore* field and the lack of marks in the cell content. A violation of the health requirements can sometimes be noted immediately:

Definition 8.6 (Coordination)**.** The state of the machine is ***coordinated*** with the current cell if it possible for them to be together in a healthy configuration. ⌟

8.2 Annotation

Configurations that have been affected by noise in a limited way only, can be "annotated", adding some information showing some of the affected parts.

We will make use of a new parameter E which approximately measures the work area needed for eliminating the damage caused by a couple of bursts. Let:

$$E = 30(Z - Z_b), \tag{17}$$

$$\text{PenetLen} = E + Z. \tag{18}$$

Definition 8.7 (Annotated configuration)**.** An ***annotated configuration*** of machine M whose cell body size is B, is a tuple

$$\mathcal{A} = (\xi, \chi, \mathcal{I}, D),$$

with the following meaning.

ξ is a configuration.

χ is a healthy configuration,

$\mathcal{I}$ is a set of intervals called ***islands***, each of size $\leqslant \beta B$. They may contain damage.

D is an interval containing the head called the ***distress area***. This is where the structure is currently being restored. It contains any island containing the head. It has size $< 5EB$.

We can obtain χ from ξ by removing damage from the islands, filling them with cells, and then by

- changing the state,
- changing the *cKind*, *cCore* and *cRec*.*Core* tracks in the islands and possibly additional $\leqslant Z$ cells within D, where *cRec*.*Core* is a set of fields used in the recovery procedure;
- changing the *cRec*.*Core* track, *cKind*, and the head position inside D.

The ***current colony*** of $\mathcal{A}$ is the base colony of χ.

We say that an interval W is the ***workspace*** of the annotated configuration $\mathcal{A}$ if it is the workspace of χ.

The following additional properties are required:

(a) At most 2 islands intersect the workspace.

There are at most 2 islands in each colony or bridge that do not intersect the workspace and the extended workspace. If there is more than one, then one of them is within a distance $\text{PenetLen} \cdot B$ from the boundary of the neighbor colony towards the base colony.

(b) If D is empty then the mode is normal.

We say that a cell is ***free*** in an annotated configuration when it is not in any island or D. The head is ***free*** when D is empty.

An annotated configuration is ***centrally consistent*** if the workspace is free. ⌟

Definition 8.8. A tuple $(\xi, \chi, \mathcal{I}, D, \mathcal{S})$, is a ***super annotated configuration*** if $(\xi, \chi, \mathcal{I}, D)$ is an annotated configuration, with the following additional properties. $\mathcal{S} \supset \mathcal{I}$ is a set of intervals of cells called ***stains***. In the base colony, either all stains but one are within a distance $E + \beta B$ to the left colony boundary, or all but one are within a distance $E + \beta B$ to the right colony boundary. This one is called the ***internal stain***. In all other colonies, all stains but one are within distance $E + \beta B$ of the boundary towards the base colony. The configuration ξ is super healthy (see the definition of health), and we obtain it by the above operations and, in addition, by changing the *cInfo* and *cState* track in the stains. ⌟

Configurations that have an annotated configuration deserve a special name.

Definition 8.9 (Admissible configuration)**.** A configuration ξ is a ***(super) admissible*** if there is a (super) annotated configuration $(\xi, \dots)$. In this case, we say that χ is a (super) healthy configuration ***satisfying*** ξ. Any change to an admissible configuration is called ***admissible***, if the resulting configuration is also admissible. ⌟

Intuitively, the definition says that a configuration is admissible on an interval, if there are not "too many" islands in that interval, and that by local changes, we can obtain a corresponding healthy configuration on the same interval. In Section 9, we will see in more detail how such a configuration can be obtained.

Definition 8.10 (Local configuration, replacement)**.** A ***local configuration on*** a (finite or infinite) interval I is given by values assigned to the cells of I, along with the following information: whether the head is to the left of, to the right of or inside I, and if it is inside, on which cell, and what is the state.

If I' is a subinterval of I, then a local configuration ξ on I clearly gives rise to a local configuration $\xi(I')$ on I' as well, called its ***subconfiguration***: If the head of ξ was in I and it was for example to the left of I', then now $\xi(I')$ just says that it is to the left, without specifying position and state.

Let ξ be a configuration and $\zeta(I)$ a local configuration that contains the head if and only if $\xi(I)$ contains the head. Then the configuration $\xi|\zeta(I)$ is obtained

by replacing ξ with ζ over the interval I, further if ξ contains the head then also replacing ξ.pos with ζ.pos and ξ.state with ζ.state. ⌟

9 Patching

In this section we show that an admissible configuration of machine M can be locally corrected. We will also show that in case the configuration is damage-free then this correction can be carried out by the machine M itself. We will deal with the elimination of damage later.

Definition 9.1. Let $\Delta = 35\beta B$. ⌟

Lemma 9.2 (Patching)**.** *Let $R = [a, b)$ be an interval containing $2E$ cells. Consider an annotated configuration $(\xi, \chi, \mathcal{I}, D)$ with the property that the ends of R are at a distance of at least $E/2$ from any colony end.*

Then it is possible to compute from $\xi.cCore(R)$ and the additional information telling which neighbor cells are adjacent, an interval

$$\hat{R} = [\hat{a}, \hat{b}) \subseteq R$$

and a local configuration $\zeta = \zeta(\hat{R})$ such that $\xi|\zeta(\hat{R})$ is healthy, and the following holds.

(a) *The states of nonempty cells of ξ can differ from the corresponding cells of ζ only in $O(\beta)$ cells, and in the interval D.*

(b) *If ξ is damage-free then the computation of ζ can be carried out by the machine M (relying only on ξ and R), using a constant number (independent of β, Q) of passes over R, and a constant number of fields containing values of size $\leqslant Q$.*

(c) *We have $\hat{a} = a$, $a + \Delta$ or $a + 3\Delta$. If $\chi.\mathrm{pos} < a + 2\Delta$ then $\hat{a} = a + 3\Delta$. The same conditions define $\hat{b}$, interchanging left and right.*

Proof. 1. Let us consolidate segments.

We will call a homogenous segment of ξ ***substantial*** if it has size at least $5\beta B$. (There are at most 4 islands in the area to be considered, so this is a segment larger than all of them combined.) We extend the notion of substantial segment somewhat, as follows: if a base colony is continued by a bridge in its drift direction, in the same sweep, then they will be said to belong to the

same substantial segment. It is convenient to introduce also a ***virtual address*** $cAddr'$ for a bridge cell, as $cAddr' = cAddr + Q \cdot cDrift$. For all other cells, $cAddr' = cAddr$. The area between two neighboring substantial segments will be called ***ambiguous***.

How large can be the ambiguous areas? There are at most 3 boundary pairs at a total size of $3B$, further 4 islands of size $\leqslant \beta B$, with non-substantial segments of sizes $< 5\beta B$ between them: this adds up to $\leqslant (34\beta + 3)B < 35\beta B = \Delta$ by Definition 9.1.

Let us call two substantial segments $[a_1, b_1)$ and $[a_2, b_2)$ ***mergeable*** if they are connected to each other with at most 35β other neighbor cells, and if $0 < cAddr'(a_2) - cAddr'(b_1 - 1) \leqslant 35\beta$.

It is easy to see that in an admissible configuration, two mergeable substantial segments can indeed be merged: one can erase the cells in the ambiguous area that cannot be added adjacently to the other ones, and replace them all with adjacent cells, having addresses growing from $cAddr(b_1)$ to $cAddr(a_2) - 1$. While doing this, *no cells outside the islands need be changed*. Machine M can recognize mergeable substantial segments and can merge them, going from left to right, one-by-one.

Let us merge also two substantial segments if they are adjacent and form a boundary of type (r2), (r3) (this would be rare), (nr1).

2. Let us restore some boundaries between substantial segments, that fall into ambiguous areas.

   Consider a boundary of type (r2): for example a left outer colony, possibly extended by an old bridge, followed by the workspace on the right: base colony or target. Then extend the base colony into the ambiguous area to the left until the left colony end is reached (erasing and replacing any cells that are in the way). After that, extend the left outer colony and bridge combination to the right, until the left end of the colony is met.

   In case of a boundary of type (r3), again we extend the target or colony side first towards the separation, and then the bridge toward the new colony end.

   Consider a boundary of type (nr1) covered by an ambiguous area: these must be two substantial segments that would be mergeable by the criterion of part 1 above, except that they differ in their sweep values by 1 (and also possibly in the kind of cells, as colony or bridge cells are converted into target cells, or

a target cells into member cells in the sweep of question). Then merge the two segments, set the sweep over the separation to be equal to the older sweep (with the necessary change in cell kinds), and place the head to the boundary of the two sweeps.

Consider a boundary of type (nr2), where a new bridge meets either an old bridge, a target, an old colony or desert. If it is meeting an old bridge or desert, then extend the new bridge end into the ambiguous area until it reaches the other bridge or a colony end, and then extend the old bridge or desert to meet it. If it is meeting an old colony or a target then first extend the colony to its maximum, then extend the new bridge to meet it.

3. Let us complete the algorithm.

In case that we started from an admissible configurations, the above operations created a sequence of substantial segments adjacent to each other, satisfying the requirements of a healthy configuration. There may still be ambiguous areas left on the ends. If there is such an area on for example the left side then we cut off Δ from the left end.

The operations we performed shifted the head of the healthy configuration by at most Δ. If by this or by the above cutting off the end, the head got closer than Δ to the end then we cut off another Δ length from that end.

It should be clear that all these operations can be accomplished by a constant number of passes on machine M.

□

Bibliography

[1] Ilir Çapuni and Peter Gács. A Turing machine resisting isolated bursts of faults. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, volume 7147 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin / Heidelberg, 2012. Full version in arXiv:1203.1335. 2

[2] Bruno Durand, Andrei Romashchenko, and Alexander Shen. Fixed-point tile sets and their applications. *Journal of Computer and System Sciences*, 78(3):731–764, 2012. 2.2

[3] Peter Gács. Reliable computation with cellular automata. *Journal of Computer System Science*, 32(1):15–78, February 1986. Conference version at STOC' 83. 2.2

[4] Peter Gács. Reliable cellular automata with self-organization. *Journal of Statistical Physics*, 103(1/2):45–267, April 2001. See also arXiv:math/0003117 [math.PR] and the proceedings of STOC '97. 2.2, 2.5

[5] G. L. Kurdyumov. An example of a nonergodic homogenous one-dimensional random medium with positive transition probabilities. *Soviet Mathematical Doklady*, 19(1):211–214, 1978. 2.2